Clase 1 — 01.10.25

#basesdedatos

📘 Acceso a Datos

🗓️ Clase 1 — 01/10/2025

🎯 Tema: Presentación de la asignatura y apuntes del tema 1

📘 Tema 1 - Gestión de Ficheros (Acceso a Datos)

1. Concepto de fichero

Un **fichero** es una unidad lógica de información almacenada en un dispositivo de memoria secundaria (disco duro, SSD, etc.).
Sus características principales son:

- Están formados por **bits**.
- Se organizan en **registros** (estructuras de datos que guardan información).
- Se identifican por **nombre y ruta** (absoluta o relativa).
- Son **no volátiles** → permanecen tras apagar el ordenador.
- Tienen una **extensión** que indica el tipo de datos (ej. `.txt`, `.jpg`, `.java`).

La **persistencia de datos** es un concepto fundamental en informática y programación que hace referencia a la **capacidad de que la información perdure en el tiempo, más allá de la ejecución de un programa o el apagado del sistema**.

En otras palabras:

- Los datos **no persistentes** existen solo en memoria (RAM) mientras el programa se ejecuta (ej. variables temporales).
- Los datos **persistentes** se almacenan en algún medio de almacenamiento estable (disco duro, SSD, base de datos, nube, etc.) y se pueden recuperar más tarde.

🔹 Tipos de ficheros

- **Ficheros estándar**:
  - Texto → almacenan caracteres legibles por humanos.
  - Binarios → almacenan datos en formato binario (no legible directamente).
- **Directorios o carpetas** → contenedores de ficheros y otros directorios.
- **Ficheros especiales** → dispositivos, sockets, enlaces, etc.

---

2. Registros

Un **registro** es una **unidad lógica de información dentro de un fichero**, que agrupa un conjunto de datos relacionados entre sí.
Se puede comparar con:

- Una **fila en una tabla de base de datos**.
- Un **objeto en memoria** dentro de un programa.

En general, un fichero se puede entender como una **colección de registros**, y estos a su vez pueden estar formados por distintos **campos** (ejemplo: nombre, apellido, edad).

🔹 Ficheros según su estructura de registros

1. **Ficheros de texto (caracteres)**
   - Guardan secuencias de caracteres codificados (ej. UTF-8).
   - Suelen organizar la información mediante **registros de longitud variable**.
   - El final de cada registro se marca con un **delimitador** como `\n` (salto de línea) o `,` (coma en CSV).
   - **Ejemplo**: un archivo `.csv` con usuarios.
   - **Acceso típico**: **secuencial**, porque se leen de principio a fin.
2. **Ficheros binarios (bytes)**
   - Almacenan la información en formato binario, optimizado para espacio y velocidad.
   - Usan habitualmente **registros de longitud fija**, lo que permite calcular directamente dónde empieza el registro *n*.
   - **Extensión típica**: `.bin`.
   - **Ejemplo**: un fichero de empleados donde cada registro ocupa exactamente 100 bytes.
   - **Ventaja**: eficiencia y posibilidad de **acceso aleatorio**.
   - **Aplicación práctica**: gestión de contraseñas (los datos pueden leerse o sobrescribirse en posiciones concretas sin recorrer todo el fichero).

🔹 Tipos de registros según la organización

1. **Registros de longitud fija**
   - Todos los registros ocupan el mismo número de bytes.
   - El tamaño del registro se define de antemano.
   - **Ventaja** → permite acceso rápido y aleatorio (`posición = n * tamañoRegistro`).
   - **Ejemplo**: fichero binario donde cada registro contiene exactamente:
     - 4 bytes para un número de ID,
     - 20 bytes para un nombre,
     - 4 bytes para una edad.
2. **Registros de longitud variable**
   - No tienen tamaño fijo, se ajustan a los datos que contienen.
   - Se separan mediante **delimitadores** (ej. salto de línea `\n`, tabulador `\t`, comas, etc.).
   - **Ventaja** → más flexibles y adaptables a datos heterogéneos.
   - **Inconveniente** → acceso más lento, ya que no se puede calcular directamente la posición de un registro.
   - **Ejemplo**:

```
Juan,Pérez,28
Ana,Díaz,33
Carmen,Sánchez,25
```

✅ **Conclusión**:

- Los **registros fijos** son más usados en **ficheros binarios** y permiten **acceso aleatorio eficiente**.

- Los **registros variables** son típicos de **ficheros de texto** y se recorren de manera **secuencial**.

---

3. Rutas y Encoding

- Una **ruta** es la dirección de un fichero dentro del sistema.
  - **Absoluta**: empieza en la raíz del sistema (`C:\...` en Windows o `/home/...` en Linux).
  - **Relativa**: depende de la carpeta de ejecución actual.

⚠️ **Problema habitual**: la diferencia entre sistemas operativos.

- Windows → usa `\`
- Linux/Mac → usan `/`

3.1 Rutas en ficheros

🔹 Problema de portabilidad

Los **separadores de directorios** varían según el sistema operativo:

- **Windows** → usa `\` (ejemplo: `C:\Users\David\archivo.txt`)
- **Linux / MacOS** → usa `/` (ejemplo: `/home/david/archivo.txt`)

⚠️ Si se escribe la ruta como un **string directo**, puede que funcione en un SO pero falle en otro:

```
// ❌ No es portable
File archivoNoSeguro = new File("carpeta/ejemplo.txt");
```

🔹 Solución con `File.separator`

La clase `File` incluye la constante `File.separator`, que adapta automáticamente el separador según la plataforma:

```
// ✅ Portable en cualquier SO
File archivo = new File("carpeta" + File.separator + "ejemplo.txt");
```

- En Windows → `"carpeta\\ejemplo.txt"`
- En Linux/Mac → `"carpeta/ejemplo.txt"`

De esta forma el programa es **portable** y no depende del sistema operativo.

🔹 Alternativas modernas (Java 7+)

La clase `Paths` de `java.nio.file` simplifica la construcción de rutas:

```
import java.nio.file.*;

Path ruta = Paths.get("carpeta", "ejemplo.txt");
```

- `Paths.get()` concatena los directorios y usa el separador correcto automáticamente.
- Más seguro y legible que concatenar strings manualmente.

✅ Resumen

- Usar rutas directas con `/` o `\` **no es seguro** entre sistemas.
- Para portabilidad:
  - `File.separator` (compatible con versiones antiguas de Java).
  - `Paths.get()` (recomendado desde Java 7).

3.2 Encoding (Codificación de caracteres)

- Es la **codificación de caracteres** (ASCII, UTF-8, ISO-8859-1, etc.).
- Afecta a la correcta lectura/escritura de acentos, ñ, emojis, etc.
- Ejemplo: escribir un fichero en UTF-8 y leerlo en ISO-8859-1 puede dar errores de símbolos.

El **encoding** define cómo se representan los caracteres como bytes en un fichero.
Es clave para que los programas puedan **leer y escribir texto de forma correcta**, sobre todo con acentos, ñ o símbolos especiales.

🔹 ASCII (American Standard Code for Information Interchange)

- Estándar de codificación creado en EE.UU. (ANSI: *American National Standards Institute*).
- Representa caracteres con **7 bits** (128 símbolos):
  - Letras inglesas (`A-Z` , `a-z`), números (`0-9`) y signos básicos.
- **Tabla extendida**:
  Usa 8 bits (256 símbolos) → incluye algunos caracteres adicionales, pero **no cubre acentos ni idiomas con alfabetos distintos**.
- Limitación: solo útil para inglés y símbolos básicos.

En realidad, cuando hablamos de **ASCII** solemos confundir dos cosas:

1. ASCII original (7 bits)

- **Nombre completo**: *American Standard Code for Information Interchange*.
- Estándar de 1963 (ANSI en EE.UU.).
- Cada carácter se representa con **7 bits** → 128 caracteres (0–127).
- Incluye:
  - Letras inglesas (`A–Z` , `a–z`).
  - Dígitos (`0–9`).
  - Símbolos básicos (`. , ; : ? !`).
  - Caracteres de control (ej. `\n` salto de línea, `\t` tabulación, `\0` null).

👉 Esta es la **tabla ASCII oficial y única**.

2. ASCII extendido (8 bits)

- **No es un estándar único oficial**, sino varias **extensiones** que aprovecharon el 8º bit (valores 128–255).
- Cada fabricante o sistema operativo lo usó para añadir caracteres propios:
  - Letras acentuadas, ñ, símbolos gráficos, etc.
- Ejemplo famoso: **ISO-8859-1 (Latin-1)** que estandariza parte de esos 128 caracteres extra para Europa occidental.

👉 Por eso a veces se habla de “dos tablas ASCII”:

- **ASCII de 7 bits (0–127)** → el estándar universal.
- **ASCII extendido de 8 bits (0–255)** → múltiples variantes, precursor de ISO-8859.

🔹 ISO-8859

- Conjunto de codificaciones reguladas por la **ISO (International Organization for Standardization)**.
- Usa **8 bits** (256 caracteres).
- Cada versión está pensada para un conjunto de idiomas:
  - **ISO-8859-1** → “Latin-1”, cubre la mayoría de lenguas de Europa occidental (incluye ñ y acentos).
  - **ISO-8859-15** → añade el símbolo del euro (€).
- Limitación: **no soporta todos los idiomas del mundo**, cada región necesita su propia variante.

🔹 Unicode

- Proyecto internacional para unificar todos los sistemas de codificación (ASCII, ISO-8859, etc.).
- Objetivo: representar **todos los caracteres de todos los idiomas** en un solo estándar.
- Incluye:
  - Letras de todos los alfabetos.
  - Símbolos matemáticos, emojis, ideogramas chinos, etc.

👉 Unicode **unifica ASCII e ISO-8859** porque:

- Los **primeros 128 caracteres de Unicode** coinciden con **ASCII**.
- Los **primeros 256 caracteres** cubren lo mismo que **ISO-8859-1**.
- A partir de ahí, se extiende a miles de símbolos.

UTF (Unicode Transformation Format)

Son **formas de codificar Unicode** en bytes.
Las más usadas son:

1. **UTF-8**
   - Variable: cada carácter ocupa entre **1 y 4 bytes**.
   - Compatible con ASCII (los 128 primeros caracteres son idénticos).
   - Muy eficiente para textos en inglés y lenguas europeas.
   - Es la codificación más usada en **web y sistemas modernos**.
2. **UTF-16**
   - Cada carácter ocupa **2 bytes** de forma estándar (y hasta 4 en casos especiales).
   - Mejor para lenguajes con alfabetos muy amplios (chino, japonés, coreano).
   - Más pesado que UTF-8 en idiomas europeos.
3. **UTF-32**
   - Cada carácter ocupa siempre **4 bytes fijos**.
   - Acceso directo a cualquier carácter (simple de manejar).
   - Muy ineficiente en espacio (triplica el tamaño respecto a UTF-8 para textos occidentales).
   - Poco usado en la práctica salvo en contextos específicos.

✅ Resumen comparativo

Codificación	Tamaño	Compatibilidad	Uso principal
ASCII	7 bits	Solo inglés	Sistemas antiguos, compatibilidad básica
ISO-8859	8 bits	Europa occidental	Textos europeos (antes de Unicode)
Unicode	Variable (UTF)	Universal	Estándar actual
UTF-8	1-4 bytes	Compatible con ASCII	Web, aplicaciones modernas
UTF-16	2 o 4 bytes	Compatible con Unicode	Idiomas asiáticos
UTF-32	4 bytes	Universal	Uso académico o técnico

---

4. Clase `File` en Java

Permite trabajar con **ficheros y directorios**.
Un objeto de tipo `File` puede representar tanto un archivo como una carpeta.

🔹 Librería `java.io`

El paquete `java.io` proporciona las clases fundamentales para realizar **operaciones de entrada y salida (Input/Output)** en Java.

Dentro de este paquete encontramos:

- Clases para **gestión de ficheros y directorios**: `File`.
- Clases para **lectura y escritura secuencial**:
  - Texto → `FileReader`, `FileWriter`.
  - Binario → `FileInputStream`, `FileOutputStream`.
- Clases para **acceso aleatorio**: `RandomAccessFile`.

🔹 Constructores principales

```
File(String nombre|ruta)
File(File ruta, String nombre)
File(String ruta, String nombre)
```

🔹 Crear directorios y ficheros con la clase `File`

Crear un directorio

Para crear un directorio en Java:

1. Crear un objeto de la clase `File`.
2. Usar el método `.mkdir()` o `.mkdirs()`.

```
// Crear un único directorio
File carpeta = new File("documentos");
boolean creado = carpeta.mkdir();

// Crear varios directorios de golpe (ruta completa)
File subcarpeta = new File("documentos/proyectos/java");
boolean creados = subcarpeta.mkdirs();
```

- `mkdir()` → crea **solo un directorio**. Falla si no existe la carpeta padre.
- `mkdirs()` → crea **todos los directorios intermedios necesarios**.

Crear un fichero

Para crear un fichero vacío:

1. Crear un objeto de la clase `File`.
2. Usar `.createNewFile()`.

```
try {
    File archivo = new File("documentos/ejemplo.txt");
    if (archivo.createNewFile()) {
        System.out.println("Fichero creado correctamente.");
    } else {
        System.out.println("El fichero ya existe.");
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Métodos más importantes de la clase `File`

Método	Descripción
`boolean mkdir()`	Crea un directorio.
`boolean mkdirs()`	Crea un directorio y todos los necesarios en la ruta.
`boolean isDirectory()`	Devuelve `true` si el objeto es un directorio.
`boolean isFile()`	Devuelve `true` si el objeto es un archivo.
`boolean delete()`	Elimina un fichero o directorio (vacío).
`boolean createNewFile()`	Crea un fichero nuevo vacío.
`void deleteOnExit()`	Marca el fichero para borrarse al terminar la JVM.
`boolean exists()`	Comprueba si el fichero/directorio existe.
`long length()`	Devuelve el tamaño en bytes del fichero.
`String getName()`	Devuelve el nombre del fichero/directorio.
`String getPath()`	Devuelve la ruta asociada.
`String getAbsolutePath()`	Devuelve la ruta absoluta.
`boolean canRead()` / `canWrite()`	Indica permisos de lectura/escritura.

Ejemplo práctico

```
File archivo = new File("documentos/ejemplo.txt");

if (!archivo.exists()) {
    archivo.createNewFile();
}

System.out.println("¿Es un fichero? " + archivo.isFile());
System.out.println("Ruta absoluta: " + archivo.getAbsolutePath());
System.out.println("Tamaño: " + archivo.length() + " bytes");
```

✅ Resumen

- Para **crear directorios** → `mkdir()` o `mkdirs()`.
- Para **crear ficheros** → `createNewFile()`.
- La clase `File` ofrece métodos para comprobar existencia, permisos, tamaño, tipo (fichero/directorio) y borrado.
- `deleteOnExit()` es útil para ficheros temporales que se deben limpiar al finalizar la ejecución.

---

5. Formas de acceso a ficheros

a) Acceso secuencial

Los datos se leen o escriben en orden, desde el principio hasta el final.
Se emplea tanto para **ficheros de texto** como **binarios**.

🔹 Manejo de excepciones en acceso secuencial

En Java, las operaciones con ficheros deben realizarse dentro de un **bloque** `try–catch`, ya que la librería `java.io` declara excepciones **predefinidas** que debemos controlar.

Ejemplo típico:

- `createNewFile()` → puede lanzar una `IOException`.
- `FileReader` → puede lanzar `FileNotFoundException`.

```
import java.io.*;

public class EjemploFichero {
    public static void main(String[] args) {
        try {
            // Crear un fichero
            File archivo = new File("documentos/ejemplo.txt");
            archivo.createNewFile();

            // Escribir en el fichero
            FileWriter escritor = new FileWriter(archivo);
            escritor.write("Hola mundo!");
            escritor.close();

            // Leer del fichero
            FileReader lector = new FileReader(archivo);
            int c;
            while ((c = lector.read()) != -1) {
                System.out.print((char) c);
            }
            lector.close();

        } catch (FileNotFoundException e) {
            System.out.println("Error: el fichero no existe -> " + e.getMessage());
        } catch (IOException e) {
            System.out.println("Error de entrada/salida -> " + e.getMessage());
        }
```

```
    }
}
```

🔹 Excepciones comunes en acceso secuencial

- **FileNotFoundException** → el fichero no existe o la ruta es incorrecta (ej. al crear un `FileReader`).
- **IOException** → error genérico de entrada/salida (ej. permisos, problemas de disco, error al cerrar un `FileWriter`).

Estas excepciones están **predefinidas en la librería `java.io`** y forman parte de la **jerarquía de excepciones de Java**.

🔹 Métodos principales en acceso secuencial

- **Clase FileReader**
  - `int read()` → lee un carácter y lo devuelve.
  - `int read(char[] buf)` → lee hasta `buf.length` caracteres en un array.
  - `int read(char[] buf, int offset, int n)` → lee `n` caracteres en `buf` desde la posición `offset`.
- **Clase FileWriter**
  - `void write(int c)` → escribe un carácter.
  - `void write(char[] buf)` → escribe un array de caracteres.
  - `void write(char[] buf, int off, int n)` → escribe parte de un array.
  - `void write(String cadena)` → escribe una cadena.
  - `void append(char c)` → añade al final del fichero.

✅ Resumen

- En **acceso secuencial** es **obligatorio** manejar excepciones con `try–catch`.
- Los métodos de `File`, `FileReader` y `FileWriter` pueden lanzar **IOException** o **FileNotFoundException**.
- Esto garantiza que el programa **no se detenga de forma inesperada** al trabajar con ficheros.
- El manejo de **errores y excepciones** será un tema central en los próximos apartados de la asignatura.

6. Operaciones sobre ficheros

Con la API de Java se pueden realizar operaciones como:

- **Crear** (`createNewFile`, `mkdir`, etc.).
- **Leer** (clases `Reader` / `InputStream`).
- **Escribir** (`Writer` / `OutputStream`).
- **Borrar** (`delete`).
- **Modificar** (sobrescribir, añadir con `append`, o editar posiciones con `RandomAccessFile`).

---

7. Comparativa rápida

Característica	Secuencial	Aleatorio
Acceso	De principio a fin	Directo a cualquier posición
Uso típico	Texto, logs, configuración	Bases de datos, registros binarios
Clases principales	`FileReader`, `FileWriter`	`RandomAccessFile`
Ventaja	Simplicidad	Flexibilidad y rapidez en acceso
Inconveniente	Ineficiente para cambios puntuales	Mayor complejidad de programación

---

✅ Resumen clave

- Un fichero es un conjunto de registros, que pueden ser texto o binario.
- Las rutas deben declararse de forma **portátil** entre SO.
- El **encoding** es fundamental para evitar errores con caracteres.
- En Java, la clase `File` gestiona ficheros/directorios.
- Hay dos tipos principales de acceso:
  - **Secuencial** → lectura lineal.
  - **Aleatorio** → acceso directo con `RandomAccessFile`.
